Theoretical Computer Science 71 (1990) 419-424
North-Holland

NOTE

A DATA STRUCTURE USEFUL FOR FINDING HAMILTONIAN CYCLES

M. CHROBAK and T. SZYMACHA
Institute of Informatics, Warsaw University, PKiN VIII p, 00-901 Warsaw, Poland

A. KRAWCZYK
Institute of Mathematics, Warsaw University, PKiN IX p, 00-901 Warsaw, Poland

Communicated by G. Mirkowska
Received March 1988
Revised September 1988

Abstract. We present a data structure, called the reversable AVL-trees, for maintaining a sequence of special operations called *reversals*. This data structure is based on balanced trees, and it yields an algorithm with complexity $O(m \log n)$, where m is the number of reversals.

1. Introduction

Some algorithms for finding Hamiltonian cycles frequently use the following operation: given a path $v_1 v_2 \cdots v_n$ and an integer $2 \leq i \leq n-2$, construct a path $v_1 v_2 \cdots v_i v_n v_{n-1} \cdots v_{i+1}$. This operation is called a *reversal*. We present a data structure, called the reversable AVL-tree (RAVL, in short), for maintaining a sequence of reversals. This data structure is based on balanced trees, and it yields an algorithm with complexity $O(m \log n)$, where m is the number of reversals.

We know of three algorithms that use the reversal operation on paths. In [8], Posa presents an algorithmic proof that graphs with n vertices and $cn \log n$ edges contain almost surely a Hamiltonian cycle, providing that c is sufficiently large (see also Karp's comments on Posa's algorithm in [4]). His result was recently improved in [3]. Their algorithm is, in a sense, optimal and its running time is $O(n^{4+\varepsilon})$. They count, apparently, $O(n)$ time for each reversal. The algorithm can also be modified to find a Hamiltonian cycle, if it exists, in polynomial expected time, or to give an approximation algorithm for the bottleneck traveling salesman problem. Another example is the algorithm suggested by Thomason [2, 10] which, for a given cubic graph G with a Hamiltonian cycle H, finds a Hamiltonian cycle in G different than H. This algorithm is in fact an effective proof of Smith's theorem (see [1, 2]). The time complexity of Thomason's algorithm has been an open problem. Poljak et al. [7] construct n-vertex graphs on which it makes cn^2 reversals. Recently the problem

was solved by Krawczyk [6], who proved that on some graphs this algorithm may make exponentially many reversals. However, the algorithm seems to be very fast on random data and we believe that its expected running time is polynomial (for appropriately defined probability distribution).

Our data structure improves the time complexity of all these three algorithms. For example, the direct application of RAVLs to the algorithm in [3] improves its time complexity to $O(n^{3+\varepsilon} \log n)$.

2. Reversable AVL-trees

We apply the idea of storing a linear list in the nodes of a balanced binary tree, as described in [5]. We choose AVL-trees for our algorithm, but any other type of balanced trees which allows $O(\log n)$ cost JOIN and SPLIT operations can also be used here (for example, 2-3 trees, red-black trees, etc. see [5, 9]). We assume that the reader is familiar with AVL-trees and the algorithms for JOIN and SPLIT operations.

To obtain a RAVL tree from an AVL we introduce a new field D in each node. This field always contains $+1$ or -1. It will turn out that a RAVL tree T in which $D(P) = +1$ for each node P represents the same linear list as the AVL obtained from T by ignoring the D-fields.

In general, we can associate with a RAVL tree T a usual AVL T^N by taking into account the D-fields in the following way. For a node P let BRD(P) be the multiple of all $D(Q)$, for the ascendants Q of P in T, including P. Informally, BRD(P) is the parity of the number of reversals of the subtree rooted at P ($+1$ = even, -1 = odd). If L = LLINK(P), R = RLINK(P) then the sons of P in T^N are also L, R but their order depends on BRD(P): if BRD(P) $= +1$ then L is left, R is right, otherwise (for BRD(P) $= -1$) L is right and R is left. Let LEFT(P) and RIGHT(P) denote such defined sons of P in T^N. It is important to remember that LEFT and RIGHT are not stored explicitly in P. And now by the linear list represented by T we will mean the list obtained by traversing T^N in inorder. Consider for example the RAVL T shown in Fig. 1. We write P^e to denote a node P with $D(P) = e$. For simplicity we identify nodes with their keys. According to the definition, T represents the list *ABCDEFGHIJKLM*.

Let REVERSE(T, P) be the procedure which reverses the list represented by T at P. If T stored the list $v_1 v_2 \cdots v_n$ and v_i = KEY(P) then after REVERSE(T, P), T represents the list $v_1 v_2 \cdots v_i v_n v_{n-1} \cdots v_{i+1}$.

How to implement REVERSE(T, P)? We can write it as

```
begin
  (T1, T2) := RSPLIT(T, P);
  D(T2) := -D(T2);
  T := RJOIN(T1, P, T2)
end,
```

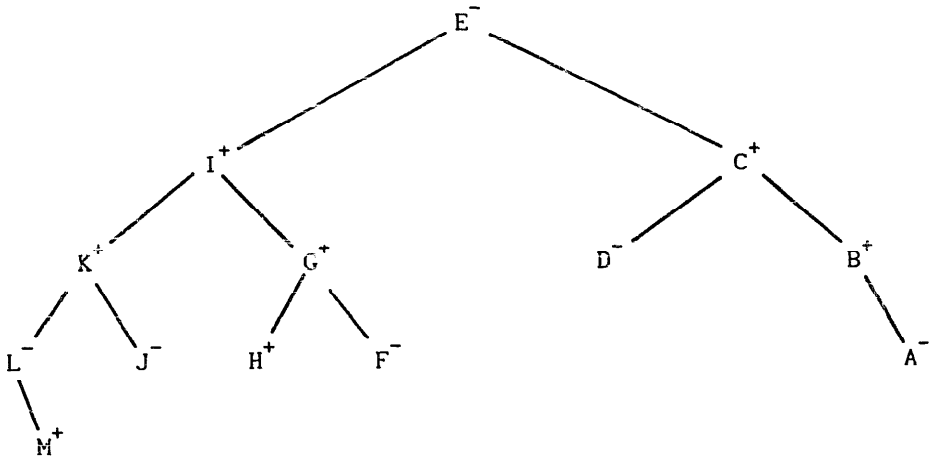

Fig. 1. The RAVL T.

where RSPLIT and RJOIN are extensions of SPLIT and JOIN, such that they do not spoil the meaning of the D-fields. After $(T_1, T_2) :=$ RSPLIT(T, P), T_1 represents the list $v_1 v_2 \cdots v_{i-1}$ and T_2 represents the list $v_{i+1} v_{i+2} \cdots v_n$; after $D(T_2) := -D(T_2)$, T_2 represents the list $v_n v_{n-1} \cdots v_{i+1}$. Finally, RJOIN$(T_1, P, T_2)$ concatenates T_1 and T_2 using P as a juncture vertex, and T will represent the list $v_1 v_2 \cdots v_i v_n v_{n-1} \cdots v_{i+1}$, as required. They also run in $O(\log n)$ time. Then the problem of implementing REVERSE reduces to the problem of implementing RSPLIT and RJOIN.

The idea of the implementation of RJOIN and RSPLIT is very simple: use the algorithm JOIN and SPLIT for AVLs except that:

(1) LEFT(P) and RIGHT(P) should be used instead of LLINK(P) and RLINK(P);

(2) wherever we cut off a tree T' rooted at P, then $D(P) := D(P) * \text{BRD}(F)$, where F is the father of P. We do the same when we attach T' to F.

In (1), BRD(P) must be correctly computed before we determine LEFT(P) and RIGHT(P). However, it is easy to see that all necessary BRD values can always be computed traversing the tree downwards, both in RSPLIT and RJOIN.

To see why (2) is necessary, imagine that BRD$(F) = -1$. Then the list represented by T' will be reversed with respect to the one it represented as a subtree of T.

It may seem at first glance that the algorithm may be difficult for programming, since we have to determine LEFT and RIGHT "links" at each step, which may increase considerably the number of cases to be considered. The simple solution of this problem is the following. Define LINK: ARRAY $[-1 \cdots +1]$ OF NODE, and let LINK$[-1]$ be the LLINK and LINK$[+1]$ the RLINK. Denoting by LINK$_P$ the LINK array stored at P we will always have

$$\text{LEFT}(P) = \text{LINK}_P[-\text{BRD}(P)],$$

$$\text{RIGHT}(P) = \text{LINK}_P[\text{BRD}(P)].$$

LINK[0] can be used for the link to the father.

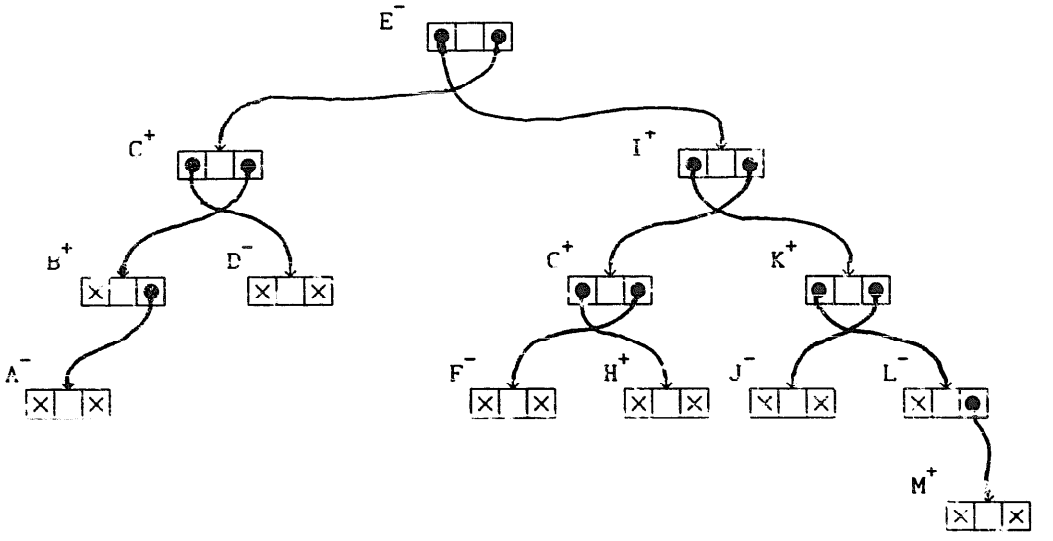

Fig. 2. The representation of T.

In Fig. 2 we show such a representation of T from Fig. 1.

3. Computational experiments

The algorithm was implemented on an IBM XT in Turbo Pascal. Its performance with two other possible algorithms is compared in Table 1. The time units are 1/100ths of a second.

The first of them, LIST, is a simple implementation in a linear two-way list, If P_i contains v_i, then REVERSE(v_i) sets the right pointer of P_i to P_n, the left pointer of P_n to P_i and exchanges the left and right pointers of $P_{i+1}, \ldots, P_{n-1}$. Of course, each reversal costs $O(n)$ time.

The second, 2LEV, is a two-layer list. We divide the whole list into blocks. Each block is of length between $\sqrt{n}$ and $2\sqrt{n}$. Each of them has a father. All fathers are also in a list and they have an additional binary field D which says whether the list of their sons is reversed or not. REVERSE(P) divides the block containing P into two parts and concatenates them with the lists to the left and right, respectively. If some of the resulting list is too long then it is divided into two equal ones. Then we change the list of the fathers similarly as in LIST, remembering that the D-fields have to be changed too. Each reversal costs $O(\sqrt{n})$ time.

4. Final remarks

It is easy to observe that after our modification all other operations on linear lists can be implemented in time $O(\log n)$. Storing additionally in each node the number of its descendants, we obtain a data structure which can handle a number of linear

Table 1

n	LIST	2LEV	RAVL
10	3.1	12.0	27.4
20	6.0	18.0	36.2
30	8.8	21.7	41.5
40	11.7	24.2	45.1
50	14.4	27.0	47.0
60	17.3	28.3	49.7
70	20.0	31.4	52.7
80	22.9	32.7	54.6
90	25.5	34.4	55.4
100	28.5	36.2	56.5
110	31.4	37.4	58.5
120	34.1	38.1	60.0
130	36.9	39.6	61.7
140	39.6	40.6	61.9
150	42.6	42.7	62.9
160	45.4	43.4	63.4
170	48.4	46.0	64.1
180	50.9	46.2	64.9
190	53.9	46.3	65.9
200	56.7	48.1	66.7
250	71.3	53.1	70.3
300	85.5	57.5	72.9
350	99.5	61.3	74.6
400	113.7	66.3	77.0
450	128.1	69.3	78.6
500	142.2	73.0	80.0
600		79.1	82.6
700		84.3	85.1
800		90.2	86.6
900		94.7	88.1
1000		97.3	89.7
1500		108.0	95.8

lists and perform the following operations in time $O(\log n)$: insert, delete, join, split, reverse, access (find the ith element in the specified list). This makes the RAVL a rather general data structure, in fact much too general for the application in which we are interested. Perhaps it would be possible to find a still more efficient data structure (at least with respect to the constant factor) capable of doing only reversals. We leave it as an open problem.

References

[1] C. Berge, *Graphs and Hypergraphs* (North-Holland, Amsterdam, 1973).

[2] B. Bollobas, *Extremal Graph Theory* (Academic Press, London, 1978).

[3] B. Bollobas, T. I. Fenner and A. M. Frieze, An algorithm for finding Hamilton cycles in a random graph, in: *Proc. 17th STOC* (1975) 359-364.

[4] R.M. Karp, The probabilistic analysis of some combinatorial search algorithms, in: J.F. Traub, ed., *Algorithms and Complexity, New Directions and Recent Results* (Academic Press, London, 1976) 1-19.
[5] D.E. Knuth, *The Art of Computer Programming*, Vol. 3, *Sorting and Searching* (Addison-Wesley, Reading, MA, 1973).
[6] A. Krawczyk, unpublished results.
[7] S. Poljak, D. Turzik and P. Pudlak, Extensions of k-subsets to $(k+1)$-subsets — existence versus constructability, *Comment. Math. Univ. Carolin.* **23** (1982) 337-349.
[8] L. Posa, Hamilton circuits in random graphs, *Discrete Math.* **14** (1976) 359-364.
[9] R.E. Tarjan, *Data Structures and Network Algorithms* (SIAM, Philadelphia, 1983).
[10] A.G. Thomason, Hamiltonian cycles and uniquely edge colourable graphs, *Ann. Discrete Math.* **3** (1978) 259-268.